AMENDMENTS TO THE CLAIMS

1. (Original) A method of optimizing address expressions within source-level code, wherein the source-level code describes the functionality of an application to be executed on a digital device, the method comprising:

inputting a first source-level code that describes the functionality of the application, the first source-level code comprising address computation code and a plurality of arrays with address expressions, wherein the address computation code or one of the address expressions has nonlinear operations; and

transforming the first source-level code into a second source-level code that describes substantially the same functionality as the first source-level code, wherein the second source-level code has fewer nonlinear operations than the first source-level code, and

wherein the digital device comprises at least one programmable instruction set processor.

2. (Original) The method of in Claim 1, wherein the nonlinear operations are selected from the group comprising: modulo operations, integer division operations, power operations and multiplication operations.

3. (Original) The method of Claim 1, wherein the transforming comprises: a first code transforming , a second code transforming and a third code transforming, the first code transforming comprising algebraic code transformations and common subexpression elimination code transformations, the second code transforming including code hoisting and the third code transforming including induction analysis.

4. (Original) The method of Claim 3, wherein first the first code transforming is executed, thereafter the second code transforming and thereafter the third code transforming.

5. (Original) The method of Claim 1, wherein the arrays are multi-dimensional arrays.

6. (Original) The method of Claim 1, wherein the first, second source-level code and intermediate source-level codes generated are represented as shared data-flow graphs.

7. (Original) The method of Claim 6, wherein the shared data-flow graphs are directed acyclic graphs with homogeneous synchronous data-flow behavior.

8. (Original) The method of claim 7, wherein the shared data-flow graph enables modeling in a single subgraph address expressions and address calculation code with substantially different iterators.

9. (Original) The method of claim 8, wherein the method uses a select operation for enabling single subgraph modeling.

10. (Original) The method of Claim 6, wherein the method time-multiplexes calculations that are related to address expressions.

11. (Original) The method of Claim 1, wherein the method is independent of the digital device architecture.

12. (Original) The method of Claim 11, wherein the method does not use detailed knowledge of the target device of the source code.

13. (Original) The method of Claim 1, wherein the method transforms the source-level codes by applying composite transformations with look-ahead local minima avoiding capabilities.

14. (Original) The method of Claim 13, wherein the composite transformations comprise a finite predetermined sequence of elementary, non-decomposable transformations, wherein each of the elementary transformations are executable if a single set of conditions related to the elementary transformation are satisfied, wherein each of the composite transformations are composed such that the composite transformation can be executed if one of a plurality of sets of conditions is satisfied.

15. (Presently Amended) The method of claim ~~15~~ 14, wherein the composite transformation transforms a first source-level code with a first cost into a second source-level code with a second cost, the second cost being lower than the first cost, and while executing the elementary transformation of which the composite transformation is composed of, intermediate source-level codes are generated, at least one of the intermediate source-level codes having a cost being larger than the first cost.

16. (Original) The method of Claim 1, wherein transforming comprises applying an algebraic transformation on at least one of the address expressions or part of the address calculation code, wherein the algebraic transformation replaces at least a part of the address expressions or part of an expression within the address computation code by another equivalent expression.

17. (Original) The method of Claim 15, wherein the execution of the composite transformations is performed on shared data-flow graph representations of the source-level codes.

18. (Original) The method of Claim 17, wherein applying the algebraic transformation increases the number of common subexpressions that are affected by nonlinear operations.

19. (Original) The method of Claim 1, wherein the transformation comprises common subexpression elimination on at least one of the address expressions or part of the address computation code, and wherein the common subexpression elimination detects a subexpression that is found within at least two expressions within either the address expressions or the address computation code and replaces the subexpression with a single variable, wherein the variable is computed via the subexpression higher in the code, the variable computation becoming part of the address computation code.

20. (Original) The method of Claim 19, wherein the common subexpression elimination decreases the amount of nonlinear operations within the source-level code.

21. (Original) The method of Claim 1, wherein at least one piece of address calculation code is located within a selected scope and transforming comprises code hoisting wherein a part of the piece of address calculation code is moved outside the scope.

22. (Original) The method of Claim 1, wherein the moving of the part of the piece of address calculation code reduces the amount of executions of nonlinear operations within the part of the piece of address calculation code.

23. (Original) The method of Claim 21, wherein code hoisting is applied across the scope of a loop construct.

24. (Original) The method of Claim 21, wherein code hoisting is applied across the scope of a conditional.

25. (Original) The method of Claim 24, wherein code hoisting is based on a range analysis.

26. (Original) The method of Claim 25, wherein code hoisting is considered for applying only for expressions with overlapping ranges.

27. (Original) The method of Claim 25, wherein the code hoisting is applied only to expressions with at least one common factor.

28. (Original) The method of Claim 25, wherein code hoisting is applied for equal expressions only when the sum of their ranges is larger than the overall range.

29. (Original) The method of Claim 25, wherein code hoisting is applied for non-equal expressions after a cost-benefit analysis that evaluates: the degree of similarity of the non-equal expressions, the degree of similarity being expressed as the amount of common subexpressions within the non-equal expressions, their costs and the cost after a potential code hoisting step.

30. (Original) The method of Claim 1 wherein modulo related algebraic transformation combined with a common subexpression elimination and code hoisting are executed iteratively.

31. (Original) The method of Claim 1, wherein a linear induction analysis based step is applied on at least one address expression, which is piece wise linear, the induction analysis step replaces the address expression with single pointer arithmetic and a single conditional.

32. (Presently Amended) The method of claim 31, wherein the piece wise linear address expression comprise at least of one of the group comprising: a modulo operation ~~and~~ or a division operation.

33. (Original) The method of Claim 32, wherein the piece wise linear address expression comprises nested operations.

34. (Original) The method of Claim 1, wherein nonlinear induction analysis is applied on at least one address expression, which is nonlinear, wherein the induction replaces the address expression with add, accumulate, constant division and constant multiplication operations or combinations thereof.

35. (Original) A system for address expression optimization of source-level code, wherein the source-level code describes the functionality of an application to be executed on a digital device, wherein the digital device comprises at least one programmable instruction set processor, the system comprising:

means for inputting a first source-level code that describes the functionality of the application, the first source-level code comprising address computation code and a plurality of arrays with address expressions, wherein at least the address computation code or one of the address expressions has nonlinear operations; and

means for transforming the first source-level code into a second source-level code that describes substantially the same functionality as the first source-level code, wherein

the second source-level code has fewer nonlinear operations than the first source-level code.

36. (Original) A system for address expression optimization of source-level code, wherein the source-level code describes the functionality of an application to be executed on a digital device, the system comprising:

an optimizing system for receiving a first source-level code that describes the functionality of the application, wherein the first source-level code has nonlinear operations, and wherein the optimizing system transforms the first source-level code into a second source-level code that has fewer nonlinear operations than the first source-level code, and wherein the digital device comprises at least one programmable instruction set processor.

37. (Original) Executable code that has been optimized by the method comprising:

transforming first source-level code that defines the operation of the executable code into a second source-level code that describes substantially the same functionality as the first source-level code, wherein the second source-level code has fewer nonlinear operations than the first source-level code, and wherein the second source-level code has been compiled for subsequent execution on a programmable instruction set processor.